JavaScript 秘密花园

wizardforcel

Published with GitBook

目錄

简介 0
对象 1
函数 2
数组 3
类型 4
核心 5
其它 6

简介

关于作者

这篇文章的作者是两位 Stack Overflow 用户, 伊沃·韦特泽尔 Ivo Wetzel（写作） 和张易江 Zhang Yi Jiang（设计）。

贡献者

- 贡献者

中文翻译

- 三生石上

此中文翻译由三生石上独立完成，博客园首发，转载请注明出处。

许可

JavaScript 秘密花园在 MIT license 许可协议下发布，并存放在 GitHub 开源社区。如果你发现错误或者打字错误，请新建一个任务单或者发一个抓取请求。 你也可以在 Stack Overflow 的 JavaScript 聊天室找到我们。

对象

对象使用和属性

JavaScript 中所有变量都可以当作对象使用，除了两个例外 `null` 和 `undefined` 。

```
false.toString(); // 'false'
[1, 2, 3].toString(); // '1,2,3'

function Foo(){}
Foo.bar = 1;
Foo.bar; // 1
```

一个常见的误解是数字的字面值（literal）不能当作对象使用。这是因为 JavaScript 解析器的一个错误， 它试图将点操作符解析为浮点数字面值的一部分。

```
2.toString(); // 出错：SyntaxError
```

有很多变通方法可以让数字的字面值看起来像对象。

```
2..toString(); // 第二个点号可以正常解析
2 .toString(); // 注意点号前面的空格
(2).toString(); // 2先被计算
```

对象作为数据类型

JavaScript 的对象可以作为哈希表使用，主要用来保存命名的键与值的对应关系。

使用对象的字面语法 - `{}` - 可以创建一个简单对象。这个新创建的对象从 `Object.prototype` 继承下面，没有任何自定义属性。

```
var foo = {}; // 一个空对象

// 一个新对象，拥有一个值为12的自定义属性'test'
var bar = {test: 12};
```

访问属性

有两种方式来访问对象的属性，点操作符或者中括号操作符。

```
var foo = {name: 'kitten'}
foo.name; // kitten
foo['name']; // kitten

var get = 'name';
foo[get]; // kitten

foo.1234; // SyntaxError
foo['1234']; // works
```

两种语法是等价的，但是中括号操作符在下面两种情况下依然有效

- 动态设置属性
- 属性名不是一个有效的变量名（译者注：比如属性名中包含空格，或者属性名是 JS 的关键词）

译者注：在 JSLint 语法检测工具中，点操作符是推荐做法。

删除属性

删除属性的唯一方法是使用 `delete` 操作符；设置属性为 `undefined` 或者 `null` 并不能真正的删除属性， 而仅仅是移除了属性和值的关联。

```
var obj = {
    bar: 1,
    foo: 2,
    baz: 3
};
obj.bar = undefined;
obj.foo = null;
delete obj.baz;

for(var i in obj) {
    if (obj.hasOwnProperty(i)) {
        console.log(i, '' + obj[i]);
    }
}
```

上面的输出结果有 `bar undefined` 和 `foo null` - 只有 `baz` 被真正的删除了，所以从输出结果中消失。

属性名的语法

```
var test = {
    'case': 'I am a keyword so I must be notated as a string',
    delete: 'I am a keyword too so me' // 出错：SyntaxError
};
```

对象的属性名可以使用字符串或者普通字符声明。但是由于 JavaScript 解析器的另一个错误设计，上面的第二种声明方式在 ECMAScript 5 之前会抛出 `SyntaxError` 的错误。

这个错误的原因是 `delete` 是 JavaScript 语言的一个关键词；因此为了在更低版本的 JavaScript 引擎下也能正常运行，必须使用字符串字面值声明方式。

原型

JavaScript 不包含传统的类继承模型，而是使用 *prototype* 原型模型。

虽然这经常被当作是 JavaScript 的缺点被提及，其实基于原型的继承模型比传统的类继承还要强大。实现传统的类继承模型是很简单，但是实现 JavaScript 中的原型继承则要困难的多。(It is for example fairly trivial to build a classic model on top of it, while the other way around is a far more difficult task.)

由于 JavaScript 是唯一一个被广泛使用的基于原型继承的语言，所以理解两种继承模式的差异是需要一定时间的。

第一个不同之处在于 JavaScript 使用原型链的继承方式。

注意: 简单的使用 `Bar.prototype = Foo.prototype` 将会导致两个对象共享相同的原型。因此，改变任意一个对象的原型都会影响到另一个对象的原型，在大多数情况下这不是希望的结果。

```
function Foo() {
    this.value = 42;
}
Foo.prototype = {
    method: function() {}
};

function Bar() {}

// 设置Bar的prototype属性为Foo的实例对象
Bar.prototype = new Foo();
Bar.prototype.foo = 'Hello World';

// 修正Bar.prototype.constructor为Bar本身
Bar.prototype.constructor = Bar;

var test = new Bar() // 创建Bar的一个新实例

// 原型链
test [Bar的实例]
    Bar.prototype [Foo的实例]
        { foo: 'Hello World' }
        Foo.prototype
            {method: ...};
            Object.prototype
                {toString: ... /* etc. */};
```

上面的例子中， `test` 对象从 `Bar.prototype` 和 `Foo.prototype` 继承下来；因此， 它能访问 `Foo` 的原型方法 `method` 。同时，它也能够访问那个定义在原型上的 `Foo` 实例属性 `value` 。 需要注意的是 `new Bar()` 不会创造出一个新的 `Foo` 实例，而是 重复使用它原型上的那个实例；因此，所有的 `Bar` 实例都会共享相同的 `value` 属性。

注意: 不要使用 `Bar.prototype = Foo` ，因为这不会执行 `Foo` 的原型，而是指向函数 `Foo` 。 因此原型链将会回溯到 `Function.prototype` 而不是 `Foo.prototype` ，因此 `method` 将不会在 Bar 的原型链上。

属性查找

当查找一个对象的属性时，JavaScript 会向上遍历原型链，直到找到给定名称的属性为止。

到查找到达原型链的顶部 - 也就是 `Object.prototype` - 但是仍然没有找到指定的属性，就会返回 undefined。

原型属性

当原型属性用来创建原型链时，可以把任何类型的值赋给它（prototype）。 然而将原子类型赋给 prototype 的操作将会被忽略。

```
function Foo() {}
Foo.prototype = 1; // 无效
```

而将对象赋值给 prototype，正如上面的例子所示，将会动态的创建原型链。

性能

如果一个属性在原型链的上端，则对于查找时间将带来不利影响。特别的，试图获取一个不存在的属性将会遍历整个原型链。

并且，当使用 `for in` 循环遍历对象的属性时，原型链上的所有属性都将被访问。

扩展内置类型的原型

一个错误特性被经常使用，那就是扩展 `Object.prototype` 或者其他内置类型的原型对象。

这种技术被称之为 monkey patching 并且会破坏封装。虽然它被广泛的应用到一些 JavaScript 类库中比如 Prototype, 但是我仍然不认为为内置类型添加一些非标准的函数是个好主意。

扩展内置类型的唯一理由是为了和新的 JavaScript 保持一致，比如 `Array.forEach` 。

译者注：这是编程领域常用的一种方式，称之为 Backport，也就是将新的补丁添加到老版本中。

总结

在写复杂的 JavaScript 应用之前，充分理解原型链继承的工作方式是每个 JavaScript 程序员必修的功课。 要提防原型链过长带来的性能问题，并知道如何通过缩短原型链来提高性能。 更进一步，绝对不要扩展内置类型的原型，除非是为了和新的 JavaScript 引擎兼容。

`hasOwnProperty` 函数

为了判断一个对象是否包含自定义属性而不是原型链上的属性， 我们需要使用继承自 `Object.prototype` 的 `hasOwnProperty` 方法。

注意: 通过判断一个属性是否 `undefined` 是不够的。 因为一个属性可能确实存在，只不过它的值被设置为 `undefined` 。

`hasOwnProperty` 是 JavaScript 中唯一一个处理属性但是不查找原型链的函数。

```
// 修改Object.prototype
Object.prototype.bar = 1;
var foo = {goo: undefined};

foo.bar; // 1
'bar' in foo; // true

foo.hasOwnProperty('bar'); // false
foo.hasOwnProperty('goo'); // true
```

只有 `hasOwnProperty` 可以给出正确和期望的结果，这在遍历对象的属性时会很有用。 没有其它方法可以用来排除原型链上的属性，而不是定义在对象自身上的属性。

`hasOwnProperty` 作为属性

JavaScript 不会保护 `hasOwnProperty` 被非法占用，因此如果一个对象碰巧存在这个属性， 就需要使用外部的 `hasOwnProperty` 函数来获取正确的结果。

```
var foo = {
    hasOwnProperty: function() {
        return false;
    },
    bar: 'Here be dragons'
};

foo.hasOwnProperty('bar'); // 总是返回 false

// 使用其它对象的 hasOwnProperty，并将其上下文设置为foo
({}).hasOwnProperty.call(foo, 'bar'); // true
```

结论

当检查对象上某个属性是否存在时， `hasOwnProperty` 是唯一可用的方法。 同时在使用 `for in` loop 遍历对象时，推荐总是使用 `hasOwnProperty` 方法， 这将会避免原型对象扩展带来的干扰。

`for in` 循环

和 `in` 操作符一样， `for in` 循环同样在查找对象属性时遍历原型链上的所有属性。

注意: `for in` 循环不会遍历那些 `enumerable` 设置为 `false` 的属性；比如数组的 `length` 属性。

```
// 修改 Object.prototype
Object.prototype.bar = 1;

var foo = {moo: 2};
for(var i in foo) {
    console.log(i); // 输出两个属性：bar 和 moo
}
```

由于不可能改变 `for in` 自身的行为，因此有必要过滤出那些不希望出现在循环体中的属性， 这可以通过 `Object.prototype` 原型上的 `hasOwnProperty` 函数来完成。

注意: 由于 `for in` 总是要遍历整个原型链，因此如果一个对象的继承层次太深的话会影响性能。

使用 `hasOwnProperty` 过滤

```
// foo 变量是上例中的
for(var i in foo) {
    if (foo.hasOwnProperty(i)) {
        console.log(i);
    }
}
```

这个版本的代码是唯一正确的写法。由于我们使用了 `hasOwnProperty`，所以这次只输出 `moo`。 如果不使用 `hasOwnProperty`，则这段代码在原生对象原型（比如 `Object.prototype`）被扩展时可能会出错。

一个广泛使用的类库 Prototype 就扩展了原生的 JavaScript 对象。 因此，当这个类库被包含在页面中时，不使用 `hasOwnProperty` 过滤的 `for in` 循环难免会出问题。

总结

推荐总是使用 `hasOwnProperty`。不要对代码运行的环境做任何假设，不要假设原生对象是否已经被扩展了。

函数

函数声明与表达式

函数是JavaScript中的一等对象，这意味着可以把函数像其它值一样传递。 一个常见的用法是把匿名函数作为回调函数传递到异步函数中。

函数声明

```
function foo() {}
```

上面的方法会在执行前被 解析(hoisted)，因此它存在于当前上下文的任意一个地方， 即使在函数定义体的上面被调用也是对的。

```
foo(); // 正常运行，因为foo在代码运行前已经被创建
function foo() {}
```

函数赋值表达式

```
var foo = function() {};
```

这个例子把一个匿名的函数赋值给变量 `foo` 。

```
foo; // 'undefined'
foo(); // 出错：TypeError
var foo = function() {};
```

由于 `var` 定义了一个声明语句，对变量 `foo` 的解析是在代码运行之前，因此 `foo` 变量在代码运行时已经被定义过了。

但是由于赋值语句只在运行时执行，因此在相应代码执行之前， `foo` 的值缺省为 undefined。

命名函数的赋值表达式

另外一个特殊的情况是将命名函数赋值给一个变量。

```
var foo = function bar() {
    bar(); // 正常运行
}
bar(); // 出错：ReferenceError
```

`bar` 函数声明外是不可见的，这是因为我们已经把函数赋值给了 `foo`；然而在 `bar` 内部依然可见。这是由于 JavaScript 的 命名处理 所致，函数名在函数内总是可见的。

注意:在IE8及IE8以下版本浏览器bar在外部也是可见的，是因为浏览器对命名函数赋值表达式进行了错误的解析，解析成两个函数 `foo` 和 `bar`

`this` 的工作原理

JavaScript 有一套完全不同于其它语言的对 `this` 的处理机制。 在五种不同的情况下 ，`this` 指向的各不相同。

全局范围内

```
this;
```

当在全部范围内使用 `this`，它将会指向全局对象。

译者注：浏览器中运行的 JavaScript 脚本，这个全局对象是 `window`。

函数调用

```
foo();
```

这里 `this` 也会指向全局对象。

ES5 注意: 在严格模式下（strict mode），不存在全局变量。 这种情况下 `this` 将会是 `undefined`。

方法调用

```
test.foo();
```

这个例子中，`this` 指向 `test` 对象。

调用构造函数

```
new foo();
```

如果函数倾向于和 `new` 关键词一块使用，则我们称这个函数是 构造函数。 在函数内部， `this` 指向新创建的对象。

显式的设置 `this`

```
function foo(a, b, c) {}

var bar = {};
foo.apply(bar, [1, 2, 3]); // 数组将会被扩展，如下所示
foo.call(bar, 1, 2, 3); // 传递到foo的参数是：a = 1, b = 2, c = 3
```

当使用 `Function.prototype` 上的 `call` 或者 `apply` 方法时，函数内的 `this` 将会被 显式设置为函数调用的第一个参数。

因此函数调用的规则在上例中已经不适用了，在 `foo` 函数内 `this` 被设置成了 `bar` 。

注意: 在对象的字面声明语法中， `this` 不能用来指向对象本身。 因此 `var obj = {me: this}` 中的 `me` 不会指向 `obj` ，因为 `this` 只可能出现在上述的五种情况中。 译者注：这个例子中，如果是在浏览器中运行， `obj.me` 等于 `window` 对象。

常见误解

尽管大部分的情况都说的过去，不过第一个规则（译者注：这里指的应该是第二个规则，也就是直接调用函数时， `this` 指向全局对象） 被认为是JavaScript语言另一个错误设计的地方，因为它从来就没有实际的用途。

```
Foo.method = function() {
    function test() {
        // this 将会被设置为全局对象（译者注：浏览器环境中也就是 window 对象
    }
    test();
}
```

一个常见的误解是 `test` 中的 `this` 将会指向 `Foo` 对象，实际上不是这样子的。

为了在 `test` 中获取对 `Foo` 对象的引用，我们需要在 `method` 函数内部创建一个局部变量指向 `Foo` 对象。

```
Foo.method = function() {
    var that = this;
    function test() {
        // 使用 that 来指向 Foo 对象
    }
    test();
}
```

`that` 只是我们随意起的名字，不过这个名字被广泛的用来指向外部的 `this` 对象。 在 闭包 一节，我们可以看到 `that` 可以作为参数传递。

方法的赋值表达式

另一个看起来奇怪的地方是函数别名，也就是将一个方法赋值给一个变量。

```
var test = someObject.methodTest;
test();
```

上例中， `test` 就像一个普通的函数被调用；因此，函数内的 `this` 将不再被指向到 `someObject` 对象。

虽然 `this` 的晚绑定特性似乎并不友好，但这确实是基于原型继承赖以生存的土壤。

```
function Foo() {}
Foo.prototype.method = function() {};

function Bar() {}
Bar.prototype = Foo.prototype;

new Bar().method();
```

当 `method` 被调用时， `this` 将会指向 `Bar` 的实例对象。

闭包和引用

闭包是 JavaScript 一个非常重要的特性，这意味着当前作用域总是能够访问外部作用域中的变量。 因为 函数 是 JavaScript 中唯一拥有自身作用域的结构，因此闭包的创建依赖于函数。

模拟私有变量

```
function Counter(start) {
    var count = start;
    return {
        increment: function() {
            count++;
        },

        get: function() {
            return count;
        }
    }
}

var foo = Counter(4);
foo.increment();
foo.get(); // 5
```

这里， `Counter` 函数返回两个闭包，函数 `increment` 和函数 `get` 。 这两个函数都维持着 对外部作用域 `Counter` 的引用，因此总可以访问此作用域内定义的变量 `count` .

为什么不可以在外部访问私有变量

因为 JavaScript 中不可以对作用域进行引用或赋值，因此没有办法在外部访问 `count` 变量。 唯一的途径就是通过那两个闭包。

```
var foo = new Counter(4);
foo.hack = function() {
    count = 1337;
};
```

上面的代码不会改变定义在 `Counter` 作用域中的 `count` 变量的值，因为 `foo.hack` 没有 定义在那个作用域内。它将会创建或者覆盖全局变量 `count` 。

循环中的闭包

一个常见的错误出现在循环中使用闭包，假设我们需要在每次循环中调用循环序号

```
for(var i = 0; i < 10; i++) {
    setTimeout(function() {
        console.log(i);
    }, 1000);
}
```

上面的代码不会输出数字 `0` 到 `9` ，而是会输出数字 `10` 十次。

当 `console.log` 被调用的时候，匿名函数保持对外部变量 `i` 的引用，此时 `for` 循环已经结束， `i` 的值被修改成了 `10`.

为了得到想要的结果，需要在每次循环中创建变量 `i` 的拷贝。

避免引用错误

为了正确的获得循环序号，最好使用 匿名包装器（译者注：其实就是我们通常说的自执行匿名函数）。

```
for(var i = 0; i < 10; i++) {
    (function(e) {
        setTimeout(function() {
            console.log(e);
        }, 1000);
    })(i);
}
```

外部的匿名函数会立即执行，并把 `i` 作为它的参数，此时函数内 `e` 变量就拥有了 `i` 的一个拷贝。

当传递给 `setTimeout` 的匿名函数执行时，它就拥有了对 `e` 的引用，而这个值是不会被循环改变的。

有另一个方法完成同样的工作，那就是从匿名包装器中返回一个函数。这和上面的代码效果一样。

```
for(var i = 0; i < 10; i++) {
    setTimeout((function(e) {
        return function() {
            console.log(e);
        }
    })(i), 1000)
}
```

`arguments` 对象

JavaScript 中每个函数内都能访问一个特别变量 `arguments`。这个变量维护着所有传递到这个函数中的参数列表。

注意: 由于 `arguments` 已经被定义为函数内的一个变量。 因此通过 `var` 关键字定义 `arguments` 或者将 `arguments` 声明为一个形式参数， 都将导致原生的 `arguments` 不会被创建。

`arguments` 变量不是一个数组（ `Array` ）。尽管在语法上它有数组相关的属性 `length` ，但它不从 `Array.prototype` 继承，实际上它是一个对象（ `Object` ）。

因此，无法对 `arguments` 变量使用标准的数组方法，比如 `push` , `pop` 或者 `slice` 。虽然使用 `for` 循环遍历也是可以的，但是为了更好的使用数组方法，最好把它转化为一个真正的数组。

转化为数组

下面的代码将会创建一个新的数组，包含所有 `arguments` 对象中的元素。

```
Array.prototype.slice.call(arguments);
```

这个转化比较慢，在性能不好的代码中不推荐这种做法。

传递参数

下面是将参数从一个函数传递到另一个函数的推荐做法。

```
function foo() {
    bar.apply(null, arguments);
}
function bar(a, b, c) {
    // 干活
}
```

另一个技巧是同时使用 `call` 和 `apply` ，创建一个快速的解绑定包装器。

```
function Foo() {}

Foo.prototype.method = function(a, b, c) {
    console.log(this, a, b, c);
};

// 创建一个解绑定的 "method"
// 输入参数为: this, arg1, arg2...argN
Foo.method = function() {

    // 结果: Foo.prototype.method.call(this, arg1, arg2... argN)
    Function.call.apply(Foo.prototype.method, arguments);
};
```

译者注：上面的 `Foo.method` 函数和下面代码的效果是一样的:

```
Foo.method = function() {
    var args = Array.prototype.slice.call(arguments);
    Foo.prototype.method.apply(args[0], args.slice(1));
};
```

自动更新

`arguments` 对象为其内部属性以及函数形式参数创建 *getter* 和 *setter* 方法。

因此，改变形参的值会影响到 `arguments` 对象的值，反之亦然。

```
function foo(a, b, c) {
    arguments[0] = 2;
    a; // 2

    b = 4;
    arguments[1]; // 4

    var d = c;
    d = 9;
    c; // 3
}
foo(1, 2, 3);
```

性能真相

不管它是否有被使用，`arguments` 对象总会被创建，除了两个特殊情况 - 作为局部变量声明和作为形式参数。

`arguments` 的 *getters* 和 *setters* 方法总会被创建；因此使用 `arguments` 对性能不会有什么影响。 除非是需要对 `arguments` 对象的属性进行多次访问。

ES5 提示: 这些 *getters* 和 *setters* 在严格模式下（strict mode）不会被创建。

译者注：在 MDC 中对 `strict mode` 模式下 `arguments` 的描述有助于我们的理解，请看下面代码：

```
// 阐述在 ES5 的严格模式下 `arguments` 的特性
function f(a) {
  "use strict";
  a = 42;
  return [a, arguments[0]];
}
var pair = f(17);
console.assert(pair[0] === 42);
console.assert(pair[1] === 17);
```

然而，的确有一种情况会显著的影响现代 JavaScript 引擎的性能。这就是使用 `arguments.callee` 。

```
function foo() {
    arguments.callee; // do something with this function object
    arguments.callee.caller; // and the calling function object
}

function bigLoop() {
    for(var i = 0; i < 100000; i++) {
        foo(); // Would normally be inlined...
    }
}
```

上面代码中， `foo` 不再是一个单纯的内联函数 inlining（译者注：这里指的是解析器可以做内联处理）， 因为它需要知道它自己和它的调用者。 这不仅抵消了内联函数带来的性能提升，而且破坏了封装，因此现在函数可能要依赖于特定的上下文。

因此强烈建议大家不要使用 `arguments.callee` 和它的属性。

ES5 提示: 在严格模式下， `arguments.callee` 会报错 `TypeError` ，因为它已经被废除了。

构造函数

JavaScript 中的构造函数和其它语言中的构造函数是不同的。 通过 `new` 关键字方式调用的函数都被认为是构造函数。

在构造函数内部 - 也就是被调用的函数内 - `this` 指向新创建的对象 `Object` 。这个新创建的对象的 `prototype` 被指向到构造函数的 `prototype` 。

如果被调用的函数没有显式的 `return` 表达式，则隐式的会返回 `this` 对象 - 也就是新创建的对象。

```
function Foo() {
    this.bla = 1;
}

Foo.prototype.test = function() {
    console.log(this.bla);
};

var test = new Foo();
```

上面代码把 `Foo` 作为构造函数调用，并设置新创建对象的 `prototype` 为 `Foo.prototype` 。

显式的 `return` 表达式将会影响返回结果，但仅限于返回的是一个对象。

```
function Bar() {
    return 2;
}
new Bar(); // 返回新创建的对象

function Test() {
    this.value = 2;

    return {
        foo: 1
    };
}
new Test(); // 返回的对象
```

译者注： `new Bar()` 返回的是新创建的对象，而不是数字的字面值 2。 因此 `new Bar().constructor === Bar` ，但是如果返回的是数字对象，结果就不同了，如下所示

```
function Bar() {
    return new Number(2);
}
new Bar().constructor === Number
```

译者注：这里得到的 `new Test()` 是函数返回的对象，而不是通过 `new` 关键字新创建的对象，因此：

```
(new Test()).value === undefined
(new Test()).foo === 1
```

如果 `new` 被遗漏了，则函数不会返回新创建的对象。

```
function Foo() {
    this.bla = 1; // 获取设置全局参数
}
Foo(); // undefined
```

虽然上例在有些情况下也能正常运行，但是由于 JavaScript 中 `this` 的工作原理， 这里的 `this` 指向全局对象。

工厂模式

为了不使用 `new` 关键字，构造函数必须显式的返回一个值。

```
function Bar() {
    var value = 1;
    return {
        method: function() {
            return value;
        }
    }
}
Bar.prototype = {
    foo: function() {}
};

new Bar();
Bar();
```

上面两种对 `Bar` 函数的调用返回的值完全相同，一个新创建的拥有 `method` 属性的对象被返回，其实这里创建了一个闭包。

还需要注意， `new Bar()` 并不会改变返回对象的原型（译者注：也就是返回对象的原型不会指向 `Bar.prototype` ）。因为构造函数的原型会被指向到刚刚创建的新对象，而这里的 `Bar` 没有把这个新对象返回（译者注：而是返回了一个包含 `method` 属性的自定义对象）。

在上面的例子中，使用或者不使用 `new` 关键字没有功能性的区别。

译者注：上面两种方式创建的对象不能访问 `Bar` 原型链上的属性，如下所示：

```
var bar1 = new Bar();
typeof(bar1.method); // "function"
typeof(bar1.foo); // "undefined"

var bar2 = Bar();
typeof(bar2.method); // "function"
typeof(bar2.foo); // "undefined"
```

通过工厂模式创建新对象

我们常听到的一条忠告是不要使用 `new` 关键字来调用函数，因为如果忘记使用它就会导致错误。

为了创建新对象，我们可以创建一个工厂方法，并且在方法内构造一个新对象。

```
function Foo() {
    var obj = {};
    obj.value = 'blub';

    var private = 2;
    obj.someMethod = function(value) {
        this.value = value;
    }

    obj.getPrivate = function() {
        return private;
    }
    return obj;
}
```

虽然上面的方式比起 `new` 的调用方式不容易出错，并且可以充分利用私有变量带来的便利， 但是随之而来的是一些不好的地方。

1. 会占用更多的内存，因为新创建的对象不能共享原型上的方法。
2. 为了实现继承，工厂方法需要从另外一个对象拷贝所有属性，或者把一个对象作为新创建对象的原型。
3. 放弃原型链仅仅是因为防止遗漏 `new` 带来的问题，这似乎和语言本身的思想相违背。

总结

虽然遗漏 `new` 关键字可能会导致问题，但这并不是放弃使用原型链的借口。 最终使用哪种方式取决于应用程序的需求，选择一种代码书写风格并坚持下去才是最重要的。

作用域与命名空间

尽管 JavaScript 支持一对花括号创建的代码段，但是并不支持块级作用域； 而仅仅支持 函数作用域。

```
function test() { // 一个作用域
    for(var i = 0; i < 10; i++) { // 不是一个作用域
        // count
    }
    console.log(i); // 10
}
```

注意: 如果不是在赋值语句中，而是在 return 表达式或者函数参数中， `{...}` 将会作为代码段解析， 而不是作为对象的字面语法解析。如果考虑到 自动分号插入，这可能会导致一些不易察觉的错误。

译者注：如果 `return` 对象的左括号和 `return` 不在一行上就会出错。

```
// 译者注：下面输出 undefined
function add(a, b) {
    return
        a + b;
}
console.log(add(1, 2));
```

JavaScript 中没有显式的命名空间定义，这就意味着所有对象都定义在一个全局共享的命名空间下面。

每次引用一个变量，JavaScript 会向上遍历整个作用域直到找到这个变量为止。 如果到达全局作用域但是这个变量仍未找到，则会抛出 `ReferenceError` 异常。

隐式的全局变量

```
// 脚本 A
foo = '42';

// 脚本 B
var foo = '42'
```

上面两段脚本效果不同。脚本 A 在全局作用域内定义了变量 `foo`，而脚本 B 在当前作用域内定义变量 `foo`。

再次强调，上面的效果完全不同，不使用 `var` 声明变量将会导致隐式的全局变量产生。

```
// 全局作用域
var foo = 42;
function test() {
    // 局部作用域
    foo = 21;
}
test();
foo; // 21
```

在函数 `test` 内不使用 `var` 关键字声明 `foo` 变量将会覆盖外部的同名变量。起初这看起来并不是大问题，但是当有成千上万行代码时，不使用 `var` 声明变量将会带来难以跟踪的 BUG。

```
// 全局作用域
var items = [/* 数组 */];
for(var i = 0; i < 10; i++) {
    subLoop();
}

function subLoop() {
    // subLoop 函数作用域
    for(i = 0; i < 10; i++) { // 没有使用 var 声明变量
        // 干活
    }
}
```

外部循环在第一次调用 `subLoop` 之后就会终止，因为 `subLoop` 覆盖了全局变量 `i`。在第二个 `for` 循环中使用 `var` 声明变量可以避免这种错误。声明变量时绝对不要遗漏 `var` 关键字，除非这就是期望的影响外部作用域的行为。

局部变量

JavaScript 中局部变量只可能通过两种方式声明，一个是作为函数参数，另一个是通过 `var` 关键字声明。

```
// 全局变量
var foo = 1;
var bar = 2;
var i = 2;

function test(i) {
    // 函数 test 内的局部作用域
    i = 5;

    var foo = 3;
    bar = 4;
}
test(10);
```

`foo` 和 `i` 是函数 `test` 内的局部变量，而对 `bar` 的赋值将会覆盖全局作用域内的同名变量。

变量声明提升（**Hoisting**）

JavaScript 会提升变量声明。这意味着 `var` 表达式和 `function` 声明都将会被提升到当前作用域的顶部。

```
bar();
var bar = function() {};
var someValue = 42;

test();
function test(data) {
    if (false) {
        goo = 1;

    } else {
        var goo = 2;
    }
    for(var i = 0; i < 100; i++) {
        var e = data[i];
    }
}
```

上面代码在运行之前将会被转化。JavaScript 将会把 `var` 表达式和 `function` 声明提升到当前作用域的顶部。

```
// var 表达式被移动到这里
var bar, someValue; // 缺省值是 'undefined'

// 函数声明也会提升
function test(data) {
    var goo, i, e; // 没有块级作用域，这些变量被移动到函数顶部
    if (false) {
        goo = 1;

    } else {
        goo = 2;
    }
    for(i = 0; i < 100; i++) {
        e = data[i];
    }
}

bar(); // 出错：TypeError，因为 bar 依然是 'undefined'
someValue = 42; // 赋值语句不会被提升规则（hoisting）影响
bar = function() {};

test();
```

没有块级作用域不仅导致 `var` 表达式被从循环内移到外部，而且使一些 `if` 表达式更难看懂。

在原来代码中，`if` 表达式看起来修改了全局变量 `goo`，实际上在提升规则被应用后，却是在修改局部变量。

如果没有提升规则（hoisting）的知识，下面的代码看起来会抛出异常 `ReferenceError`。

```
// 检查 SomeImportantThing 是否已经被初始化
if (!SomeImportantThing) {
    var SomeImportantThing = {};
}
```

实际上，上面的代码正常运行，因为 `var` 表达式会被提升到全局作用域的顶部。

```
var SomeImportantThing;

// 其它一些代码，可能会初始化 SomeImportantThing，也可能不会

// 检查是否已经被初始化
if (!SomeImportantThing) {
    SomeImportantThing = {};
}
```

译者注：在 Nettuts+ 网站有一篇介绍 hoisting 的文章，其中的代码很有启发性。

```
// 译者注：来自 Nettuts+ 的一段代码，生动的阐述了 JavaScript 中变量声明提升
var myvar = 'my value';

(function() {
    alert(myvar); // undefined
    var myvar = 'local value';
})();
```

名称解析顺序

JavaScript 中的所有作用域，包括全局作用域，都有一个特别的名称 `this` 指向当前对象。

函数作用域内也有默认的变量 `arguments`，其中包含了传递到函数中的参数。

比如，当访问函数内的 `foo` 变量时，JavaScript 会按照下面顺序查找：

1. 当前作用域内是否有 `var foo` 的定义。
2. 函数形式参数是否有使用 `foo` 名称的。
3. 函数自身是否叫做 `foo`。
4. 回溯到上一级作用域，然后从 **#1** 重新开始。

注意: 自定义 `arguments` 参数将会阻止原生的 `arguments` 对象的创建。

命名空间

只有一个全局作用域导致的常见错误是命名冲突。在 JavaScript中，这可以通过 匿名包装器 轻松解决。

```
(function() {
    // 函数创建一个命名空间

    window.foo = function() {
        // 对外公开的函数，创建了闭包
    };

})(); // 立即执行此匿名函数
```

匿名函数被认为是 表达式；因此为了可调用性，它们首先会被执行。

```
( // 小括号内的函数首先被执行
function() {}
) // 并且返回函数对象
() // 调用上面的执行结果，也就是函数对象
```

有一些其他的调用函数表达式的方法，比如下面的两种方式语法不同，但是效果一模一样。

```
// 另外两种方式
+function(){}();
(function(){}());
```

结论

推荐使用匿名包装器（译者注：也就是自执行的匿名函数）来创建命名空间。这样不仅可以防止命名冲突， 而且有利于程序的模块化。

另外，使用全局变量被认为是不好的习惯。这样的代码容易产生错误并且维护成本较高。

数组

数组遍历与属性

虽然在 JavaScript 中数组是对象，但是没有好的理由去使用 `for in` 循环 遍历数组。 相反，有一些好的理由不去使用 `for in` 遍历数组。

注意: JavaScript 中数组不是 关联数组。 JavaScript 中只有对象 来管理键值的对应关系。但是关联数组是保持顺序的，而对象不是。

由于 `for in` 循环会枚举原型链上的所有属性，唯一过滤这些属性的方式是使用 `hasOwnProperty` 函数， 因此会比普通的 `for` 循环慢上好多倍。

遍历

为了达到遍历数组的最佳性能，推荐使用经典的 `for` 循环。

```
var list = [1, 2, 3, 4, 5, ...... 100000000];
for(var i = 0, l = list.length; i < l; i++) {
    console.log(list[i]);
}
```

上面代码有一个处理，就是通过 `l = list.length` 来缓存数组的长度。

虽然 `length` 是数组的一个属性，但是在每次循环中访问它还是有性能开销。 可能最新的 JavaScript 引擎在这点上做了优化，但是我们没法保证自己的代码是否运行在这些最近的引擎之上。

实际上，不使用缓存数组长度的方式比缓存版本要慢很多。

`length` 属性

`length` 属性的 *getter* 方式会简单的返回数组的长度，而 *setter* 方式会截断数组。

```
var foo = [1, 2, 3, 4, 5, 6];
foo.length = 3;
foo; // [1, 2, 3]

foo.length = 6;
foo; // [1, 2, 3]
```

译者注：在 Firebug 中查看此时 `foo` 的值是： `[1, 2, 3, undefined, undefined, undefined]` 但是这个结果并不准确，如果你在 Chrome 的控制台查看 `foo` 的结果，你会发现是这样的： `[1, 2, 3]` 因为在 JavaScript 中 `undefined` 是一个变量，注意是变量不是关键字，因此上面两个结果的意义是完全不相同的。

```
// 译者注：为了验证，我们来执行下面代码，看序号 5 是否存在于 foo 中。
5 in foo; // 不管在 Firebug 或者 Chrome 都返回 false
foo[5] = undefined;
5 in foo; // 不管在 Firebug 或者 Chrome 都返回 true
```

为 `length` 设置一个更小的值会截断数组，但是增大 `length` 属性值不会对数组产生影响。

结论

为了更好的性能，推荐使用普通的 `for` 循环并缓存数组的 `length` 属性。使用 `for in` 遍历数组被认为是不好的代码习惯并倾向于产生错误和导致性能问题。

`Array` 构造函数

由于 `Array` 的构造函数在如何处理参数时有点模棱两可，因此总是推荐使用数组的字面语法 - `[]` - 来创建数组。

```
[1, 2, 3]; // 结果: [1, 2, 3]
new Array(1, 2, 3); // 结果: [1, 2, 3]

[3]; // 结果: [3]
new Array(3); // 结果: []
new Array('3') // 结果: ['3']

// 译者注：因此下面的代码将会使人很迷惑
new Array(3, 4, 5); // 结果: [3, 4, 5]
new Array(3) // 结果: []，此数组长度为 3
```

译者注：这里的模棱两可指的是数组的两种构造函数语法

由于只有一个参数传递到构造函数中（译者注：指的是 `new Array(3);` 这种调用方式），并且这个参数是数字，构造函数会返回一个 `length` 属性被设置为此参数的空数组。需要特别注意的是，此时只有 `length` 属性被设置，真正的数组并没有生成。

译者注：在 Firebug 中，你会看到 `[undefined, undefined, undefined]` ，这其实是不对的。在上一节有详细的分析。

```
var arr = new Array(3);
arr[1]; // undefined
1 in arr; // false, 数组还没有生成
```

这种优先于设置数组长度属性的做法只在少数几种情况下有用，比如需要循环字符串，可以避免 `for` 循环的麻烦。

```
new Array(count + 1).join(stringToRepeat);
```

译者注： `new Array(3).join('#')` 将会返回 `##`

结论

应该尽量避免使用数组构造函数创建新数组。推荐使用数组的字面语法。它们更加短小和简洁，因此增加了代码的可读性。

类型

相等与比较

JavaScript 有两种方式判断两个值是否相等。

等于操作符

等于操作符由两个等号组成：`==`

JavaScript 是弱类型语言，这就意味着，等于操作符会为了比较两个值而进行强制类型转换。

```
""           ==   "0"           // false
0            ==   ""            // true
0            ==   "0"           // true
false        ==   "false"       // false
false        ==   "0"           // true
false        ==   undefined     // false
false        ==   null          // false
null         ==   undefined     // true
" \t\r\n"    ==   0             // true
```

上面的表格展示了强制类型转换，这也是使用 `==` 被广泛认为是不好编程习惯的主要原因， 由于它的复杂转换规则，会导致难以跟踪的问题。

此外，强制类型转换也会带来性能消耗，比如一个字符串为了和一个数字进行比较，必须事先被强制转换为数字。

严格等于操作符

严格等于操作符由三个等号组成：`===`

不像普通的等于操作符，严格等于操作符不会进行强制类型转换。

```
""           ===   "0"           // false
0            ===   ""            // false
0            ===   "0"           // false
false        ===   "false"       // false
false        ===   "0"           // false
false        ===   undefined     // false
false        ===   null          // false
null         ===   undefined     // false
" \t\r\n"    ===   0             // false
```

上面的结果更加清晰并有利于代码的分析。如果两个操作数类型不同就肯定不相等也有助于性能的提升。

比较对象

虽然 `==` 和 `===` 操作符都是等于操作符，但是当其中有一个操作数为对象时，行为就不同了。

```
{} === {};                   // false
new String('foo') === 'foo'; // false
new Number(10) === 10;       // false
var foo = {};
foo === foo;                 // true
```

这里等于操作符比较的不是值是否相等，而是是否属于同一个身份；也就是说，只有对象的同一个实例才被认为是相等的。 这有点像 Python 中的 `is` 和 C 中的指针比较。

注意:为了更直观的看到`==`和`===`的区别,可以参见JavaScript Equality Table

结论

强烈推荐使用严格等于操作符。如果类型需要转换，应该在比较之前显式的转换，而不是使用语言本身复杂的强制转换规则。

`typeof` 操作符

`typeof` 操作符（和 `instanceof` 一起）或许是 JavaScript 中最大的设计缺陷， 因为几乎不可能从它们那里得到想要的结果。

尽管 `instanceof` 还有一些极少数的应用场景， `typeof` 只有一个实际的应用（译者注：这个实际应用是用来检测一个对象是否已经定义或者是否已经赋值），而这个应用却不是用来检查对象的类型。

注意: 由于 `typeof` 也可以像函数的语法被调用，比如 `typeof(obj)` ，但这并不是一个函数调用。 那两个小括号只是用来计算一个表达式的值，这个返回值会作为 `typeof` 操作符的一个操作数。 实际上不存在名为 `typeof` 的函数。

JavaScript 类型表格

```
Value               Class      Type
-------------------------------------
"foo"               String     string
new String("foo")   String     object
1.2                 Number     number
new Number(1.2)     Number     object
true                Boolean    boolean
new Boolean(true)   Boolean    object
new Date()          Date       object
new Error()         Error      object
[1,2,3]             Array      object
new Array(1, 2, 3)  Array      object
new Function("")    Function   function
/abc/g              RegExp     object (function in Nitro/V8)
new RegExp("meow")  RegExp     object (function in Nitro/V8)
{}                  Object     object
new Object()        Object     object
```

上面表格中，*Type* 一列表示 `typeof` 操作符的运算结果。可以看到，这个值在大多数情况下都返回 "object"。

Class 一列表示对象的内部属性 `[[Class]]` 的值。

JavaScript 标准文档中定义: `[[Class]]` 的值只可能是下面字符串中的一个： `Arguments` , `Array` , `Boolean` , `Date` , `Error` , `Function` , `JSON` , `Math` , `Number` , `Object` , `RegExp` , `String` .

为了获取对象的 `[[Class]]` ，我们需要使用定义在 `Object.prototype` 上的方法 `toString` 。

对象的类定义

JavaScript 标准文档只给出了一种获取 `[[Class]]` 值的方法，那就是使用 `Object.prototype.toString` 。

```
function is(type, obj) {
    var clas = Object.prototype.toString.call(obj).slice(8, -1);
    return obj !== undefined && obj !== null && clas === type;
}

is('String', 'test'); // true
is('String', new String('test')); // true
```

上面例子中， `Object.prototype.toString` 方法被调用，this 被设置为了需要获取 `[[Class]]` 值的对象。

译者注： `Object.prototype.toString` 返回一种标准格式字符串，所以上例可以通过 `slice` 截取指定位置的字符串，如下所示：

```
Object.prototype.toString.call([])    // "[object Array]"
Object.prototype.toString.call({})    // "[object Object]"
Object.prototype.toString.call(2)    // "[object Number]"
```

ES5 提示: 在 ECMAScript 5 中，为了方便，对 `null` 和 `undefined` 调用 `Object.prototype.toString` 方法， 其返回值由 `Object` 变成了 `Null` 和 `Undefined` 。

译者注：这种变化可以从 IE8 和 Firefox 4 中看出区别，如下所示：

```
// IE8
Object.prototype.toString.call(null)    // "[object Object]"
Object.prototype.toString.call(undefined)    // "[object Object]"

// Firefox 4
Object.prototype.toString.call(null)    // "[object Null]"
Object.prototype.toString.call(undefined)    // "[object Undefined]
```

测试为定义变量

```
typeof foo !== 'undefined'
```

上面代码会检测 `foo` 是否已经定义；如果没有定义而直接使用会导致 `ReferenceError` 的异常。 这是 `typeof` 唯一有用的地方。

结论

为了检测一个对象的类型，强烈推荐使用 `Object.prototype.toString` 方法；因为这是唯一一个可依赖的方式。正如上面表格所示， `typeof` 的一些返回值在标准文档中并未定义， 因此不同的引擎实现可能不同。

除非为了检测一个变量是否已经定义，我们应尽量避免使用 `typeof` 操作符。

`instanceof` 操作符

`instanceof` 操作符用来比较两个操作数的构造函数。只有在比较自定义的对象时才有意义。 如果用来比较内置类型，将会和 `typeof` 操作符 一样用处不大。

比较自定义对象

```
function Foo() {}
function Bar() {}
Bar.prototype = new Foo();

new Bar() instanceof Bar; // true
new Bar() instanceof Foo; // true

// 如果仅仅设置 Bar.prototype 为函数 Foo 本身，而不是 Foo 构造函数的一个实
Bar.prototype = Foo;
new Bar() instanceof Foo; // false
```

`instanceof` 比较内置类型

```
new String('foo') instanceof String; // true
new String('foo') instanceof Object; // true

'foo' instanceof String; // false
'foo' instanceof Object; // false
```

有一点需要注意，`instanceof` 用来比较属于不同 JavaScript 上下文的对象（比如，浏览器中不同的文档结构）时将会出错， 因为它们的构造函数不会是同一个对象。

结论

`instanceof` 操作符应该仅仅用来比较来自同一个 JavaScript 上下文的自定义对象。 正如 `typeof` 操作符一样，任何其它的用法都应该是避免的。

类型转换

JavaScript 是弱类型语言，所以会在任何可能的情况下应用强制类型转换。

```
// 下面的比较结果是：true
new Number(10) == 10; // Number.toString() 返回的字符串被再次转换为数字

10 == '10';           // 字符串被转换为数字
10 == '+10 ';         // 同上
10 == '010';          // 同上
isNaN(null) == false; // null 被转换为数字 0
                      // 0 当然不是一个 NaN（译者注：否定之否定）

// 下面的比较结果是：false
10 == 010;
10 == '-10';
```

ES5 提示: 以 `0` 开头的数字字面值会被作为八进制数字解析。 而在 ECMAScript 5 严格模式下，这个特性被移除了。

为了避免上面复杂的强制类型转换，强烈推荐使用严格的等于操作符。 虽然这可以避免大部分的问题，但 JavaScript 的弱类型系统仍然会导致一些其它问题。

内置类型的构造函数

内置类型（比如 `Number` 和 `String`）的构造函数在被调用时，使用或者不使用 `new` 的结果完全不同。

```
new Number(10) === 10;     // False, 对象与数字的比较
Number(10) === 10;         // True, 数字与数字的比较
new Number(10) + 0 === 10; // True, 由于隐式的类型转换
```

使用内置类型 `Number` 作为构造函数将会创建一个新的 `Number` 对象， 而在不使用 `new` 关键字的 `Number` 函数更像是一个数字转换器。

另外，在比较中引入对象的字面值将会导致更加复杂的强制类型转换。

最好的选择是把要比较的值显式的转换为三种可能的类型之一。

转换为字符串

```
'' + 10 === '10'; // true
```

将一个值加上空字符串可以轻松转换为字符串类型。

转换为数字

```
+'10' === 10; // true
```

使用一元的加号操作符，可以把字符串转换为数字。

译者注：字符串转换为数字的常用方法：

```
+'010' === 10
Number('010') === 10
parseInt('010', 10) === 10  // 用来转换为整数

+'010.2' === 10.2
Number('010.2') === 10.2
parseInt('010.2', 10) === 10
```

转换为布尔型

通过使用 否 操作符两次，可以把一个值转换为布尔型。

```
!!'foo';   // true
!!'';      // false
!!'0';     // true
!!'1';     // true
!!'-1'     // true
!!{};      // true
!!true;    // true
```

核心

为什么不要使用 `eval`

`eval` 函数会在当前作用域中执行一段 JavaScript 代码字符串。

```
var foo = 1;
function test() {
    var foo = 2;
    eval('foo = 3');
    return foo;
}
test(); // 3
foo; // 1
```

但是 `eval` 只在被直接调用并且调用函数就是 `eval` 本身时，才在当前作用域中执行。

```
var foo = 1;
function test() {
    var foo = 2;
    var bar = eval;
    bar('foo = 3');
    return foo;
}
test(); // 2
foo; // 3
```

译者注：上面的代码等价于在全局作用域中调用 `eval`，和下面两种写法效果一样：

```
// 写法一：直接调用全局作用域下的 foo 变量
var foo = 1;
function test() {
    var foo = 2;
    window.foo = 3;
    return foo;
}
test(); // 2
foo; // 3

// 写法二：使用 call 函数修改 eval 执行的上下文为全局作用域
var foo = 1;
function test() {
    var foo = 2;
    eval.call(window, 'foo = 3');
    return foo;
}
test(); // 2
foo; // 3
```

在任何情况下我们都应该避免使用 `eval` 函数。99.9% 使用 `eval` 的场景都有不使用 `eval` 的解决方案。

伪装的 `eval`

定时函数 `setTimeout` 和 `setInterval` 都可以接受字符串作为它们的第一个参数。 这个字符串总是在全局作用域中执行，因此 `eval` 在这种情况下没有被直接调用。

安全问题

`eval` 也存在安全问题，因为它会执行任意传给它的代码， 在代码字符串未知或者是来自一个不信任的源时，绝对不要使用 `eval` 函数。

结论

绝对不要使用 `eval`，任何使用它的代码都会在它的工作方式，性能和安全性方面受到质疑。 如果一些情况必须使用到 `eval` 才能正常工作，首先它的设计会受到质疑，这不应该是首选的解决方案， 一个更好的不使用 `eval` 的解决方案应该得到充分考虑并优先采用。

`undefined` 和 `null`

JavaScript 有两个表示‘空’的值，其中比较有用的是 `undefined`。

`undefined` 的值

`undefined` 是一个值为 `undefined` 的类型。

这个语言也定义了一个全局变量，它的值是 `undefined` ，这个变量也被称为 `undefined` 。 但是这个变量不是一个常量，也不是一个关键字。这意味着它的值可以轻易被覆盖。

ES5 提示: 在 ECMAScript 5 的严格模式下， `undefined` 不再是 可写的了。 但是它的名称仍然可以被隐藏，比如定义一个函数名为 `undefined` 。

下面的情况会返回 `undefined` 值：

- 访问未修改的全局变量 `undefined` 。
- 由于没有定义 `return` 表达式的函数隐式返回。
- `return` 表达式没有显式的返回任何内容。
- 访问不存在的属性。
- 函数参数没有被显式的传递值。
- 任何被设置为 `undefined` 值的变量。

处理 `undefined` 值的改变

由于全局变量 `undefined` 只是保存了 `undefined` 类型实际值的副本， 因此对它赋新值不会改变类型 `undefined` 的值。

然而，为了方便其它变量和 `undefined` 做比较，我们需要事先获取类型 `undefined` 的值。

为了避免可能对 `undefined` 值的改变，一个常用的技巧是使用一个传递到匿名包装器的额外参数。 在调用时，这个参数不会获取任何值。

```
var undefined = 123;
(function(something, foo, undefined) {
    // 局部作用域里的 undefined 变量重新获得了 `undefined` 值

})('Hello World', 42);
```

另外一种达到相同目的方法是在函数内使用变量声明。

```
var undefined = 123;
(function(something, foo) {
    var undefined;
    ...

})('Hello World', 42);
```

这里唯一的区别是，在压缩后并且函数内没有其它需要使用 `var` 声明变量的情况下，这个版本的代码会多出 4 个字节的代码。

译者注：这里有点绕口，其实很简单。如果此函数内没有其它需要声明的变量，那么 `var` 总共 4 个字符（包含一个空白字符）就是专门为 `undefined` 变量准备的，相比上个例子多出了 4 个字节。

`null` 的用处

JavaScript 中的 `undefined` 的使用场景类似于其它语言中的 *null*，实际上 JavaScript 中的 `null` 是另外一种数据类型。

它在 JavaScript 内部有一些使用场景（比如声明原型链的终结 `Foo.prototype = null`），但是大多数情况下都可以使用 `undefined` 来代替。

自动分号插入

尽管 JavaScript 有 C 的代码风格，但是它不强制要求在代码中使用分号，实际上可以省略它们。

JavaScript 不是一个没有分号的语言，恰恰相反上它需要分号来就解析源代码。 因此 JavaScript 解析器在遇到由于缺少分号导致的解析错误时，会自动在源代码中插入分号。

```
var foo = function() {
} // 解析错误，分号丢失
test()
```

自动插入分号，解析器重新解析。

```
var foo = function() {
}; // 没有错误，解析继续
test()
```

自动的分号插入被认为是 JavaScript 语言最大的设计缺陷之一，因为它能改变代码的行为。

工作原理

下面的代码没有分号，因此解析器需要自己判断需要在哪些地方插入分号。

```
(function(window, undefined) {
    function test(options) {
        log('testing!')

        (options.list || []).forEach(function(i) {

        })

        options.value.test(
            'long string to pass here',
            'and another long string to pass'
        )

        return
        {
            foo: function() {}
        }
    }
    window.test = test

})(window)

(function(window) {
    window.someLibrary = {}
})(window)
```

下面是解析器"猜测"的结果。

```
(function(window, undefined) {
    function test(options) {

        // 没有插入分号，两行被合并为一行
        log('testing!')(options.list || []).forEach(function(i) {

        }); // <- 插入分号

        options.value.test(
            'long string to pass here',
            'and another long string to pass'
        ); // <- 插入分号

        return; // <- 插入分号, 改变了 return 表达式的行为
        { // 作为一个代码段处理
            foo: function() {}
        }; // <- 插入分号
    }
    window.test = test; // <- 插入分号

// 两行又被合并了
})(window)(function(window) {
    window.someLibrary = {}; // <- 插入分号
})(window); //<- 插入分号
```

注意: JavaScript 不能正确的处理 `return` 表达式紧跟换行符的情况， 虽然这不能算是自动分号插入的错误，但这确实是一种不希望的副作用。

解析器显著改变了上面代码的行为，在另外一些情况下也会做出错误的处理。

前置括号

在前置括号的情况下，解析器不会自动插入分号。

```
log('testing!')
(options.list || []).forEach(function(i) {})
```

上面代码被解析器转换为一行。

```
log('testing!')(options.list || []).forEach(function(i) {})
```

`log` 函数的执行结果极大可能不是函数；这种情况下就会出现 `TypeError` 的错误，详细错误信息可能是 `undefined is not a function` 。

结论

建议绝对不要省略分号，同时也提倡将花括号和相应的表达式放在一行， 对于只有一行代码的 `if` 或者 `else` 表达式，也不应该省略花括号。 这些良好的编程习惯不仅可以提到代码的一致性，而且可以防止解析器改变代码行为的错误处理。

其它

`setTimeout` 和 `setInterval`

由于 JavaScript 是异步的，可以使用 `setTimeout` 和 `setInterval` 来计划执行函数。

注意: 定时处理不是 ECMAScript 的标准，它们在 DOM (文档对象模型) 被实现。

```
function foo() {}
var id = setTimeout(foo, 1000); // 返回一个大于零的数字
```

当 `setTimeout` 被调用时，它会返回一个 ID 标识并且计划在将来大约 1000 毫秒后调用 `foo` 函数。 `foo` 函数只会被执行一次。

基于 JavaScript 引擎的计时策略，以及本质上的单线程运行方式，所以其它代码的运行可能会阻塞此线程。 因此没法确保函数会在 `setTimeout` 指定的时刻被调用。

作为第一个参数的函数将会在全局作用域中执行，因此函数内的 `this` 将会指向这个全局对象。

```
function Foo() {
    this.value = 42;
    this.method = function() {
        // this 指向全局对象
        console.log(this.value); // 输出：undefined
    };
    setTimeout(this.method, 500);
}
new Foo();
```

注意: `setTimeout` 的第一个参数是函数对象，一个常犯的错误是这样的 `setTimeout(foo(), 1000)` ， 这里回调函数是 `foo` 的返回值，而不是 `foo` 本身。 大部分情况下，这是一个潜在的错误，因为如果函数返回 `undefined` ， `setTimeout` 也不会报错。

`setInterval` 的堆调用

`setTimeout` 只会执行回调函数一次，不过 `setInterval` - 正如名字建议的 - 会每隔 `X` 毫秒执行函数一次。 但是却不鼓励使用这个函数。

当回调函数的执行被阻塞时， `setInterval` 仍然会发布更多的回调指令。在很小的定时间隔情况下，这会导致回调函数被堆积起来。

```
function foo(){
    // 阻塞执行 1 秒
}
setInterval(foo, 100);
```

上面代码中， `foo` 会执行一次随后被阻塞了一秒钟。

在 `foo` 被阻塞的时候， `setInterval` 仍然在组织将来对回调函数的调用。 因此，当第一次 `foo` 函数调用结束时，已经有 **10** 次函数调用在等待执行。

处理可能的阻塞调用

最简单也是最容易控制的方案，是在回调函数内部使用 `setTimeout` 函数。

```
function foo(){
    // 阻塞执行 1 秒
    setTimeout(foo, 100);
}
foo();
```

这样不仅封装了 `setTimeout` 回调函数，而且阻止了调用指令的堆积，可以有更多的控制。 `foo` 函数现在可以控制是否继续执行还是终止执行。

手工清空定时器

可以通过将定时时产生的 ID 标识传递给 `clearTimeout` 或者 `clearInterval` 函数来清除定时， 至于使用哪个函数取决于调用的时候使用的是 `setTimeout` 还是 `setInterval` 。

```
var id = setTimeout(foo, 1000);
clearTimeout(id);
```

清除所有定时器

由于没有内置的清除所有定时器的方法，可以采用一种暴力的方式来达到这一目的。

```
// 清空"所有"的定时器
for(var i = 1; i < 1000; i++) {
    clearTimeout(i);
}
```

可能还有些定时器不会在上面代码中被清除（译者注：如果定时器调用时返回的 ID 值大于 1000），因此我们可以事先保存所有的定时器 ID，然后一把清除。

隐藏使用 `eval`

`setTimeout` 和 `setInterval` 也接受第一个参数为字符串的情况。这个特性绝对不要使用，因为它在内部使用了 `eval`。

注意: 由于定时器函数不是 ECMAScript 的标准，如何解析字符串参数在不同的 JavaScript 引擎实现中可能不同。事实上，微软的 JScript 会使用 `Function` 构造函数来代替 `eval` 的使用。

```
function foo() {
    // 将会被调用
}

function bar() {
    function foo() {
        // 不会被调用
    }
    setTimeout('foo()', 1000);
}
bar();
```

由于 `eval` 在这种情况下不是被直接调用，因此传递到 `setTimeout` 的字符串会自全局作用域中执行；因此，上面的回调函数使用的不是定义在 `bar` 作用域中的局部变量 `foo`。

建议不要在调用定时器函数时，为了向回调函数传递参数而使用字符串的形式。

```
function foo(a, b, c) {}

// 不要这样做
setTimeout('foo(1,2, 3)', 1000)

// 可以使用匿名函数完成相同功能
setTimeout(function() {
    foo(1, 2, 3);
}, 1000)
```

注意: 虽然也可以使用这样的语法 `setTimeout(foo, 1000, 1, 2, 3)` ，但是不推荐这么做，因为在使用对象的属性方法时可能会出错。（译者注：这里说的是属性方法内， `this` 的指向错误）

结论

绝对不要使用字符串作为 `setTimeout` 或者 `setInterval` 的第一个参数， 这么写的代码明显质量很差。当需要向回调函数传递参数时，可以创建一个匿名函数，在函数内执行真实的回调函数。

另外，应该避免使用 `setInterval` ，因为它的定时执行不会被 JavaScript 阻塞。